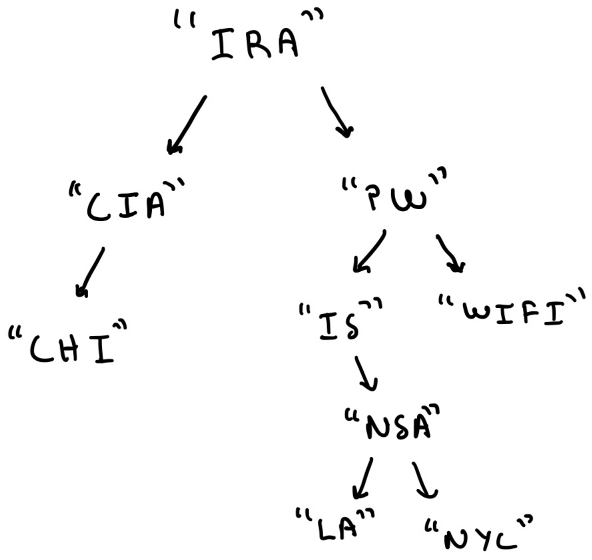
Task 3.1 With the key “LA CHI CIA,” the message “NSA IS CIA IS IRA” is encrypted as: “LA PW IRA CIA WIFI”.

Task 3.2 With the key “$w_{k_0} ... w_{k_{l-1}}$,” a mathematical expression for encrypting is: $w_{c_i} = w_{(m_i + k_{i \,\%\, l}) \,\%\, n}$.

Task 3.3 With the key “$w_{k_0} ... w_{k_{l-1}}$,” a mathematical expression for decrypting is: $w_{m_i} = w_{(c_i - k_{i \,\%\, l} + n) \,\%\, n}$.

Task 5.1



Task 5.2

```
int num_trees(int n) {
    if (n == 0) return 1;
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += num_trees(i) * num_trees(n - 1 - i);
    }
    return total;
}
```

Task 5.3 I would include the following additional fields in struct node:

1. int nodes;

   - Keeps track of the total number of nodes in the subtree corresponding with the particular node. For example, a node with two children which are both leaves would have value 3.

- Maintain the value by incrementing the field in each node passed during a traversal when calling insert and initializing each new node with value 1, and decrementing the field in each node passed during a traversal when calling delete.

2. int minPrice;
   - Keeps track of the minimum price of the nodes in the subtree corresponding with the particular node. For example, a node of price 100 with two children which are both leaves and have prices 50 and 150 would have value 50.
   - Maintain the value by updating the field in each node passed during a traversal when calling insert with the minimum of minPrice and the price of the new node. When calling delete, if minPrice matches the price of the node being deleted, track the minPrice of the other children as the BST is traversed and update minPrice accordingly.

3. int maxPrice;
   - Keeps track of the maximum price of the nodes in the subtree corresponding with the particular node. For example, a node of price 100 with two children which are both leaves and have prices 50 and 150 would have value 150.
   - Maintain the value by updating the field in each node passed during a traversal when calling insert with the maximum of maxPrice and the price of the new node. When calling delete, if maxPrice matches the price of the node being deleted, track the maxPrice of the other children as the BST is traversed and update maxPrice accordingly.

Task 5.4

```
tree *insert(tree *T, char *addr, int dist, int price) {
    if (T == NULL) {
        // leaf; make a new node
        T = malloc(sizeof(struct node));
        T->distance = dist;
        T->price = price;
        T->address = malloc(strlen(addr) + 1);
        strcpy(T->address, addr);
        T->minPrice = price;
        T->maxPrice = price;
        T->nodes = 1;
    }
    else if (dist < T->distance) {
        // insert into left subtree
        (T->nodes)++;
        T->left = insert(T->left, addr, dist, price);
        if (price < T->minPrice) T->minPrice = price;
        if (price > T->maxPrice) T->maxPrice = price;
    }
    else if (dist > T->distance) {
        // insert into right subtree
        (T->nodes)++;
        T->right = insert(T->right, addr, dist, price);
        if (price < T->minPrice) T->minPrice = price;
        if (price > T->maxPrice) T->maxPrice = price;
    }
    else {
        // duplicate; not possible
        printf("Error: Properties with same distance encountered.\n");
    }
 return T;
}
```

The insert function has logarithmic runtime because we perform a constant number of operations at each level of the BST, and we go down one level each time we recurse. Since the BST is assumed to be balanced, the height is logarithmic in the number of nodes. Hence, our average number of operations is a constant times log(n), and the asymptotic runtime of insert is O(log n).

Task 5.5 For the range query, we use two helper functions:

- int greater (T, d)
  - if (T->distance > d) return greater(T->left) + (T->right->nodes) + 1

  - else return greater(T->right)

- int less (T, d)
  - if (T->distance < d) return less(T->right) + (T->left->nodes) + 1
  - else return less(T->left)

Since both greater and less do a constant amount of work at each level, and each recursion drops down one level, if we assume the BST is balanced, our average number of operations is a constant times log(n), and the asymptotic runtimes of greater and less are O(log n). We now define range as follows:

- int range (T, d1, d2)
  - return T->nodes - less(T, d1) - greater(T, d2)

Hence range takes the total number of nodes, a constant-time operation, and subtracts those outside of the range by calling two logarithmic functions. Our average number of operations is a constant times log(n), and the asymptotic runtime of range is O(log n).

Task 5.6 For the price bound query, we use two helper functions, greaterDistance and lowerDistance. greaterDistance will find the minimum and maximum prices of properties with distances greater than the lower bound for distances. Similarly, lowerDistance will find the minimum and maximum prices of properties with distances less than the upper bound for distances. price_bound will make sure there are properties satisfying the distance requirement by first calling range, and returning NULL if it returns 0. price_bound will then combine these helper functions by initializing arrays for them to fill (the first entry, representing the minimum, with INT_MAX, and the second entry, representing the maximum, with INT_MIN), and then taking the maximum of the first entries (minimum prices) of the arrays they fill and the minimum of the second entries (maximum prices) of the arrays they fill and returning these in a new array.

I will discuss greaterDistance, and the implementation of lower bound is analogous. greaterDistance(T, d, range) accepts a tree, a distance, and an integer array to fill with the minimum and maximum prices. We look at the current node.

- If its distance is greater than d, we update the first entry of the array with the minimum of its current value and the minPrice at the current node, and we update the second entry of the array with the maximum of its current value and the maxPrice at the current node. We then recurse on left subtree, threading through the same distance and array.

- If its distance is less than d, we simply recurse on the right subtree, because all the entries in the left subtree are less than d.

This algorithm runs in logarithmic time, because it performs constant time operations and calls greaterDistance and lowerDistance. greaterDistance and lowerDistance both perform constant work at each level before recursing. If we assume the BST is balanced, our average number of operations is a constant times log(n), and the asymptotic runtimes of greaterDistance and lowerDistance are O(log n), and hence the asymptotic runtime of price_bound is O(log n).

Task 6.1

$$\begin{tikzcd}
& & & & 3 \arrow[dll] \arrow[drr] & & & \\
& & 6 \arrow[dl] \arrow[dr] & & & & 5 \arrow[dl] \arrow[dr] & \\
& 8 \arrow[dl] \arrow[dr] & & 9 & & 19 & & 10 \\
15 & & 22 & & & & &
\end{tikzcd}$$

Task 6.2

$$\begin{tikzcd}
& & & & 5 \arrow[dll] \arrow[drr] & & & \\
& & 6 \arrow[dl] \arrow[dr] & & & & 10 \arrow[dl] \arrow[dr] & \\
& 8 \arrow[dl] & & 9 & & 19 & & 22 \\
15 & & & & & & &
\end{tikzcd}$$

Task 6.3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	5	6	10	8	9	19	22	15							

Task 6.4

```
bool heap_search(heap *H, int p, int idx) {
    if (H[idx] > p) {
        return false;
    }
    else if (H[idx] == p) {
        return true;
    }
    else if (H[idx] < p) {
        return heap_search(H, p, idx * 2) || heap_search(H, p, idx * 2 + 1);
    }
}
```

Task 6.5 I would use a data structure consisting of two binary heaps, a min heap and a max heap (just a min heap with the order property flipped). Both heaps are implemented with an integer array of priorities, an integer capacity, and an integer size. The min heap uses the standard implementation of insert (as insertMin) and deleteMin, and it contains all the elements greater than the median. Similarly, the max heap uses analogous implementations of insert (as insertMax) and deleteMax, and it contains all the elements less than the median. We maintain as an invariant that the sizes of the two heaps differ by at most 1.

findMedian: Due to the aforementioned size invariant, if the size of the max heap is greater than the size of the min heap, the median is the element at the root of the max heap, and we return the root (index 1 in the array) of the max heap. Similarly, if the size of the min heap is greater than or equal to the size of the max heap, we call return the root (index 1 in the array) of the min heap. Since we are only accessing an array element and checking a constant-time conditional, the asymptotic runtime of findMedian if O(1).

deleteMedian: Due to the aforementioned size invariant, if the size of the max heap is greater than the size of the min heap, the median is the element at the root of the max heap, and we call and return deleteMax on the max heap. Similarly, if the size of the min heap is greater than or equal to the size of the max heap, we call and return deleteMin on the min heap. We then decrement the size of the appropriate heap. Note that calling deleteMedian necessarily maintains our size invariant, because we never remove an element from the smaller heap. Since deleteMin and deleteMax both run in constant time, the asymptotic runtime of deleteMedian is O(log n).

insert: To insert an element, we first compare the element to the median, which is found using the method detailed in deleteMedian. If it is greater than the median, we call insertMin on the min heap to insert the element. Otherwise, we call insertMax on the max heap to insert the element. We increment the size of the appropriate heap, and then we check to make sure our size invariant is preserved. If the sizes of the tables differ by at most 1, we return from the function. Otherwise, we call deleteMin or deleteMax on the larger heap, and then call insertMin or insertMax on the opposite heap with the removed node. Since the sizes could not have differed by more than 2, our size invariant is now satisfied and we return from the function. Since we call insertMin or insertMax a maximum of two times, and these functions both run in logarithmic time, the asymptotic runtime of insert is O(log n).